Le langage Ada – 15 : conteneurs

Yves Bailly

EN DEUX MOTS Pour Ada comme pour tout langage de programmation, les types fondamentaux deviennent rapidement insuffisants. Des structures plus sophistiquées comme les listes chaînées ou les dictionnaires sont indispensables pour tout développement d'envergure. La norme Ada2005 enrichit la bibliothèque standard de telles structures.

C'était en effet un manque important du langage Ada95 (c'est-à-dire, le langage Ada tel que défini par la norme de 1995). Chacun réinventait alors dans son coin sa propre roue, ou bien utilisait des ensembles de paquetages tels que Booch [1] ou Simple Components [2]. Situation évidemment loin d'être idéale. Aussi, la dernière version de la norme Ada (2005) inclut-elle désormais un ensemble de structures de données répondant à ces besoins.

Organisation

L'ensemble des conteneurs Ada prend racine dans le paquetage `Ada.Containers`. Celui-ci ne définit rien d'autre qu'une paire de types : `Hash_Type` pour représenter une valeur de hachage (issue d'une fonction de hachage) et `Count_Type` pour représenter le nombre d'éléments dans un conteneur. Ces deux types dépendent de l'implémentation, toutefois `Count_Type` est un type entier modulaire.

On retrouve les grandes familles usuelles des conteneurs, présentées dans le tableau suivant avec leur correspondance en langage C++ (dans le tableau, `AC` représente `Ada.Containers`).

	Ada		C++	
	Type	Paquetage	Classe	En-tête
Liste doublement chaînée	`List`	`AC.Doubly_Linked_Lists`	`std::list`	`<list>`
Vecteur (tableau linéaire)	`Vector`	`AC.Vectors`	`std::vector`	`<vector>`
Ensemble ordonné	`Set`	`AC.Ordered_Sets`	`std::set`	`<set>`
Ensemble non ordonné	`Set`	`AC.Hashed_Sets`	`std::unordered_set`	`<unordered_set>`
Dictionnaire ordonné	`Map`	`AC.Ordered_Maps`	`std::map`	`<map>`
Dictionnaire non ordonné	`Map`	`AC.Hashed_Maps`	`std::unordered_map`	`<unordered_map>`

Dans les structures ordonnées, les éléments sont rangés selon une relation de type « inférieur à » (généralement dans un arbre), tandis que les structures non ordonnées reposent sur une fonction de hachage pour placer les éléments dans une table.

Il convient de noter que les classes `unordered_set` et `unordered_map`, déclarées dans l'espace de nommage `std::tr1`, ne sont pas encore normalisées pour le C++.

Voyons maintenant comment tout cela s'utilise.

Le type Vector

Pour commencer, nous allons nous pencher sur le type `Vector`, qui représente un tableau linéaire dynamique. Les fonctionnalités présentées sont généralement également disponibles pour le type `List` (liste doublement chaînée), à l'exception de tout ce qui concerne l'indexation des éléments, la notion d'indice n'existant pas pour une liste.

La déclaration du paquetage est celle-ci :

```
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors is
...
    type Vector is tagged private;
...
end  Ada.Containers.Vectors;
```

Il s'agit donc d'un paquetage générique, ce qui est bien le moins que l'on puisse attendre d'un tel outil. Les paramètres génériques sont :

- `Index_Type` représente le type utilisé comme indice pour ce vecteur ; cela peut être n'importe quel type entier : comme les tableaux prédéfinis, il est possible d'indexer les éléments par des nombres négatifs, le premier indice n'étant pas forcément 1 (ou 0) ;
- `Element_Type` est le type des éléments à stocker ; ce type ne doit pas être limité, ce qui signifie qu'il doit être possible d'affecter des valeurs entre elles par l'affectation `:=` ;

- Enfin, la fonction "=" est celle qui sera utilisée pour comparer deux éléments, notamment pour la recherche ; la présence de is <> à la fin de la déclaration du paramètre signale que celui-ci est optionnel, l'égalité standard étant utilisée par défaut.

Remarquez que le type Vector est un type qualifié par tagged : on peut donc lui appliquer toutes les techniques de la programmation objet, en particulier le dériver en un nouveau type, l'étendre en lui ajoutant des données ou des opérations.

Voyons immédiatement un exemple simple d'utilisation :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Vectors;
procedure Vector_1 is
   package Int_Vects is
      new Ada.Containers.Vectors(Index_Type => Positive,
                                 Element_Type => Positive);
```

Il est tout d'abord nécessaire d'instancier le paquetage générique. Nous créons ici un nouveau paquetage, nommé Int_Vects, permettant de déclarer des vecteurs d'entiers positifs indexés par des entiers positifs (pour mémoire, le type Positive est défini comme un sous-type de Integer limité à l'intervalle 1..Integer'Last).

```
   vect: Int_Vects.Vector;
begin
   vect.Set_Length(10);
```

Nous pouvons alors déclarer une variable de notre « nouveau » type Vector, défini dans notre « nouveau » paquetage Int_Vects. L'opération Set_Length() définit le nombre d'éléments contenus dans le vecteur, ce nombre pouvant être modifié par la suite.

Le type Vector distingue la *taille* de la *capacité*. La taille représente le nombre d'éléments effectivement contenus dans le conteneur. Elle est fixée par Set_Length(), donnée par Length() et Is_Empty() retourne un booléen à True ou False selon que cette taille est nulle ou non.

Par contre, la capacité représente le nombre d'éléments qui *pourraient* être stockés sans provoquer une nouvelle allocation de mémoire, autrement dit, la taille maximale que l'on peut demander sans que cela donne lieu à une allocation de mémoire supplémentaire (et incidemment, la potentielle duplication des données existantes dans la nouvelle zone mémoire). Elle est fixée par Reserve_Capacity() et donnée par Capacity(). Si v est un vecteur, on a donc toujours v.Length() <= v.Capacity(). Notez, par ailleurs, que l'on peut avoir un vecteur vide (v.Length() retourne 0, ou v.Is_Empty() retourne True) sans que la capacité soit nulle.

```
   for i in 1..10
   loop
      vect.Replace_Element(i, i**2);
   end loop;
```

La boucle précédente remplit le vecteur – ou plutôt, elle remplace les éléments existants par d'autres. En effet, l'appel à Set_Length() a implicitement créé 10 éléments non initialisés. Notre tableau est donc déjà plein, sauf que les valeurs nous sont inconnues. Pour placer un élément à un indice i, on utilise Replace_Element() avec en paramètres l'indice en question et la valeur voulue. Remarquez la précision de la sémantique utilisée : on ne *stocke* pas une valeur dans le vecteur, on *remplace* une valeur existante par une autre.

```
   for i in 1..10
   loop
      Put(Positive'Image(vect.Element(i)));
   end loop;
   New_Line;
end Vector_1;
```

Enfin, on affiche les valeurs contenues dans le vecteur, l'élément situé à un indice donné étant obtenu par l'opération Element().

Opérateurs sur les vecteurs

Le type Vector étant simplement privé, il est possible d'affecter entre eux des vecteurs, même de tailles différentes (mais contenant des éléments de même type), par l'opérateur :=. Par ailleurs, deux vecteurs peuvent être comparés pour l'égalité au moyen de l'opérateur =, ce qui revient à comparer les éléments des vecteurs.

Plus amusant est l'opérateur de concaténation &, que nous connaissons déjà sur les tableaux et les chaînes de caractères. Par exemple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Vectors;
procedure Vector_2 is
   package Int_Vects is
      new Ada.Containers.Vectors(Index_Type => Positive,
                                 Element_Type => Positive);
   use type Int_Vects.Vector;
```

Le début est similaire à l'exemple précédent. On a simplement ajouté une clause use type, ce qui permet de rendre visibles toutes les opérations associées à un type donné (donc les opérateurs, ce qui nous intéresse ici) sans pour autant rendre visible tout le paquetage dans lequel est déclaré ce type.

```
   procedure Put_Line(v: in Int_Vects.Vector) is
   begin
      for i in 1..v.Length
      loop
         Put(Positive'Image(v.Element(Positive(i))) & " ");
      end loop;
      New_Line;
   end Put_Line;
```

Pour nous simplifier la tâche, on crée une procédure permettant d'afficher le contenu d'un vecteur. Le transtypage (en rouge) est nécessaire, car l'indice de boucle i est implicitement de type Ada.Containers.Count_Type (le type retourné par l'opération Length()), alors que l'opération Element() attend un paramètre de type Positive (en fait, du type générique formel Index_Type).

```
   v1: Int_Vects.Vector;
   v2: Int_Vects.Vector;
begin
   v1 := 1 & 2;
   Put_Line(v1);
```

Premier exemple, la concaténation de deux valeurs produit un vecteur. Naturellement, cela ne fonctionne que si le type des deux opérandes est celui des éléments du vecteur, et que le récepteur du résultat (ici, l'opération d'affectation) est bien du type vecteur ainsi créé.

On pourrait enchaîner ainsi les concaténations, par exemple quelque chose comme :

```
Put_Line(10 & 20 & 30 & 40);
```

mais ce n'est pas recommandé, étant donné que la création puis la destruction des résultats intermédiaires peuvent s'avérer coûteuses.

```
   v2 := v1 & 3;
   Put_Line(v2);
   v1 := 4 & v1;
   Put_Line(v1);
```

Il est également possible d'étendre un vecteur en le concaténant avec une valeur. La première ligne est équivalente à :

```
v2 := v1;
v2.Append(3);
```

tandis que la deuxième (qui modifie `v1` en y plaçant le résultat) est équivalente à :

```
v1.Prepend(4);
```

Les opérations `Append()` (ajouter à la fin) et `Prepend()` (ajouter au début) prennent un paramètre supplémentaire optionnel indiquant le nombre de copies de la valeur ajoutée.

```
   v2 := v2 & v1;
   Put_Line(v2);
end Vector_2;
```

Enfin, il est également possible de concaténer deux vecteurs pour en produire un nouveau, comme le montre le code précédent. Cette commande est équivalente à :

```
v2.Append(v1);
```

Les opérations `Append()` et `Prepend()` peuvent en effet recevoir un vecteur en entrée, plutôt qu'un simple élément.

Ce programme affiche tout simplement :

```
 1  2
 1  2  3
 4  1  2
 1  2  3  4  1  2
```

Il existe également une série d'opérations `Insert()` qui permettent d'insérer une valeur au milieu d'un vecteur, en redimensionnant celui-ci au besoin. Mais notez que ce genre d'action est généralement coûteux, car cela implique un déplacement en mémoire des éléments se trouvant après l'élément inséré. Si vous devez effectuer fréquemment de telles insertions (ou suppressions avec `Delete()`), considérez plutôt le type `List`.

Les curseurs

Nous avons jusqu'ici manipulé les éléments par le biais de leur indice. Mais il existe un autre moyen d'y accéder, disponible sur tous les types de conteneurs : le *curseur*, représenté par le type privé `Cursor`. Il s'agit là de l'équivalent des itérateurs de la bibliothèque standard du langage C++.

Voici un exemple déterminant quelques nombres premiers par l'algorithme du crible d'Ératosthène s'appuyant sur une liste (le programme est prodigieusement inefficace, mais c'est pour l'exemple) :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Doubly_Linked_Lists;
procedure Sieve is
   package Int_Lists is
      new Ada.Containers.Doubly_Linked_Lists
         (Element_Type => Positive);
   use type Int_Lists.Cursor;
```

Pour commencer, on réalise l'instanciation du paquetage `Doubly_Linked_Lists`, en ne donnant cette fois que le type des éléments (il n'y a pas d'indice et la fonction de comparaison est celle par défaut). La clause `use type` nous facilitera l'accès au type `Cursor` de ce nouveau paquetage.

```
   procedure Put_Line(l: in Int_Lists.List) is
      c: Int_Lists.Cursor;
   begin
      Put("(");
      c := l.First;
      while c /= Int_Lists.No_Element
       loop
         Put(Positive'Image(Int_Lists.Element(c)));
         c := Int_Lists.Next(c);
      end loop;
      Put_Line(" )");
   end Put_Line;
```

Commençons par une simple procédure affichant la liste. Un curseur est tout d'abord déclaré pour parcourir celle-ci. On lui affecte le résultat de l'opération `First()`, qui donne un curseur « pointant sur » le premier élément de la liste. Puis, on boucle, jusqu'à ce que notre curseur reçoive la valeur `No_Element`.

Cette valeur constante, de type `Cursor`, est déclarée dans tous les paquetages de conteneurs. Elle représente un curseur ne pointant sur rien. Dès qu'un curseur reçoit cette valeur, il ne peut tout simplement plus être utilisé, à moins de lui affecter une nouvelle valeur. On a donc une sémantique assez différente de celle du C++, où on peut avoir un itérateur pointant juste un cran au-delà des bornes d'un conteneur.

L'élément associé au curseur est donné par la fonction `Element()`. Remarquez que comme le type `Cursor` n'est pas qualifié par `tagged`, on ne peut pas utiliser la notation pointée sur une instance de ce type : il est donc nécessaire de préfixer `Element()` par le nom du paquetage concerné.

Le passage à l'élément suivant se fait par la fonction `Next()`, qui retourne un curseur pointant sur l'élément suivant ou `No_Element` si on est déjà à la fin de la liste. Cette fonction existe également sous la forme d'une procédure modifiant son paramètre (passé en mode `in out`).

Voyons maintenant comment nous pouvons manipuler la liste :

```
procedure Prime(l: in out Int_Lists.List;
                p:    out Positive) is
    first: Positive;
    last : Positive;
  begin
    first := l.First_Element;
    last := l.Last_Element;
```

Les opérations `First_Element()` et `Last_Element()` retournent respectivement le premier et le dernier élément du conteneur. À ne pas confondre avec `First()` et `Last()`, qui retournent des *curseurs* pointant sur ces éléments.

```
    if first*first > last
    then
      l.Delete_First;
```

`Delete_First()` a simplement pour effet de retirer le premier élément de la liste, qui se trouve ainsi plus courte. L'opération symétrique `Delete_Last()` retire le dernier élément. Ces deux opérations sont également disponibles sur les vecteurs, mais, sur ceux-ci, `Delete_First()` est relativement coûteuse.

```
    else
      declare
        curs: Int_Lists.Cursor;
        next: Int_Lists.Cursor;
        elem: Positive;
      begin
        curs := l.First;
        while curs /= Int_Lists.No_Element
        loop
          next := Int_Lists.Next(curs);
          elem := Int_Lists.Element(curs);
          if (elem mod first) = 0
          then
            l.Delete(curs);
          end if;
          curs := next;
        end loop;
      end;
```

Ce qui précède est le cœur du crible : on recherche tous les multiples du premier élément et on les retire de la liste. On utilise pour cela l'opération `Delete()`, qui prend en paramètre (outre le conteneur concerné) un curseur pointant sur le premier élément à supprimer et éventuellement un nombre d'éléments à supprimer. À l'issu de cet appel, le curseur vaut `No_Element` : dans notre cas, nous devons donc en sauvegarder une copie pour avancer dans la liste.

```
    end if;
    p := first;
  end Prime;
  lst: Int_Lists.List;
  p  : Positive;
begin
  for i in 1..25
  loop
    lst.Append(2*i+1);
  end loop;
  while not lst.Is_Empty
  loop
    Put_Line(lst);
    Prime(lst, p);
    Put_Line(Positive'Image(p));
  end loop;
end Sieve;
```

Le programme se termine en affichant simplement le nombre premier trouvé et l'état de la liste à mesure que le crible avance. On obtient l'affichage suivant :

```
( 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 )
 3
( 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 )
 5
( 7 11 13 17 19 23 29 31 37 41 43 47 49 )
 7
( 11 13 17 19 23 29 31 37 41 43 47 )
 11
( 13 17 19 23 29 31 37 41 43 47 )
 13
( 17 19 23 29 31 37 41 43 47 )
 17
( 19 23 29 31 37 41 43 47 )
 19
( 23 29 31 37 41 43 47 )
 23
( 29 31 37 41 43 47 )
 29
( 31 37 41 43 47 )
 31
( 37 41 43 47 )
 37
( 41 43 47 )
 41
( 43 47 )
 43
( 47 )
 47
```

Ce qui illustre bien les réductions successives de la liste.

Les types ensemblistes

Les types `Hashed_Set` et `Ordered_Set` représentent le concept mathématique d'ensemble, c'est-à-dire une collection de valeurs sans duplication. La différence essentielle est que le premier repose sur une fonction de hachage pour organiser ses éléments, tandis que le second repose sur une relation d'ordre entre les éléments. Par exemple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Hashed_Sets;
with Ada.Containers.Ordered_Sets;
procedure Set_1 is
  function Int_Hash(i: in Integer)
    return Ada.Containers.Hash_Type is
  begin
    return Ada.Containers.Hash_Type(abs i);
  end Int_Hash;
  package Int_Hash_Sets is
    new Ada.Containers.Hashed_Sets
       (Element_Type        => Integer,
        Hash                => Int_Hash,
        Equivalent_Elements => "=");
  use type Int_Hash_Sets.Cursor;
```

Première instanciation, celle du paquetage `Hashed_Sets` pour contenir des entiers. La

fonction de hachage consiste simplement à renvoyer la valeur absolue d'une valeur (paramètre générique `Hash` du paquetage), tandis que l'équivalence entre deux valeurs est obtenue par la simple égalité (paramètre générique `Equivalent_Elements`).

```
procedure Put_Line(hs: in Int_Hash_Sets.Set) is
   curs: Int_Hash_Sets.Cursor;
begin
...
end Put_Line;
```

La procédure précédente affiche le contenu d'un ensemble. Nous ne la détaillerons pas, elle est tout à fait similaire à celle vue pour les listes.

```
package Int_Ordered_Sets is
   new Ada.Containers.Ordered_Sets
      (Element_Type => Integer);
use type Int_Ordered_Sets.Cursor;

procedure Put_Line(hs: in Int_Ordered_Sets.Set) is
   curs: Int_Ordered_Sets.Cursor;
begin
...
end Put_Line;
```

Ensuite on instancie le paquetage `Ordered_Sets`, toujours pour contenir des entiers. Cette fois, il suffit de donner le type des éléments, la fonction de comparaison étant par défaut l'opérateur `"<"` prédéfini.

Voyons maintenant ce que cela donne, en insérant les mêmes valeurs dans le même ordre dans chacun des deux conteneurs :

```
   hashed : Int_Hash_Sets.Set;
   ordered: Int_Ordered_Sets.Set;
begin
   hashed.Insert(-1);
   hashed.Insert(-2);
   hashed.Insert(1);
   hashed.Insert(2);
   Put_Line(hashed);
   ordered.Insert(-1);
   ordered.Insert(-2);
   ordered.Insert(1);
   ordered.Insert(2);
   Put_Line(ordered);
end Set_1;
```

L'opération `Insert()` permet d'ajouter un élément à un ensemble. Résultat de l'exécution :

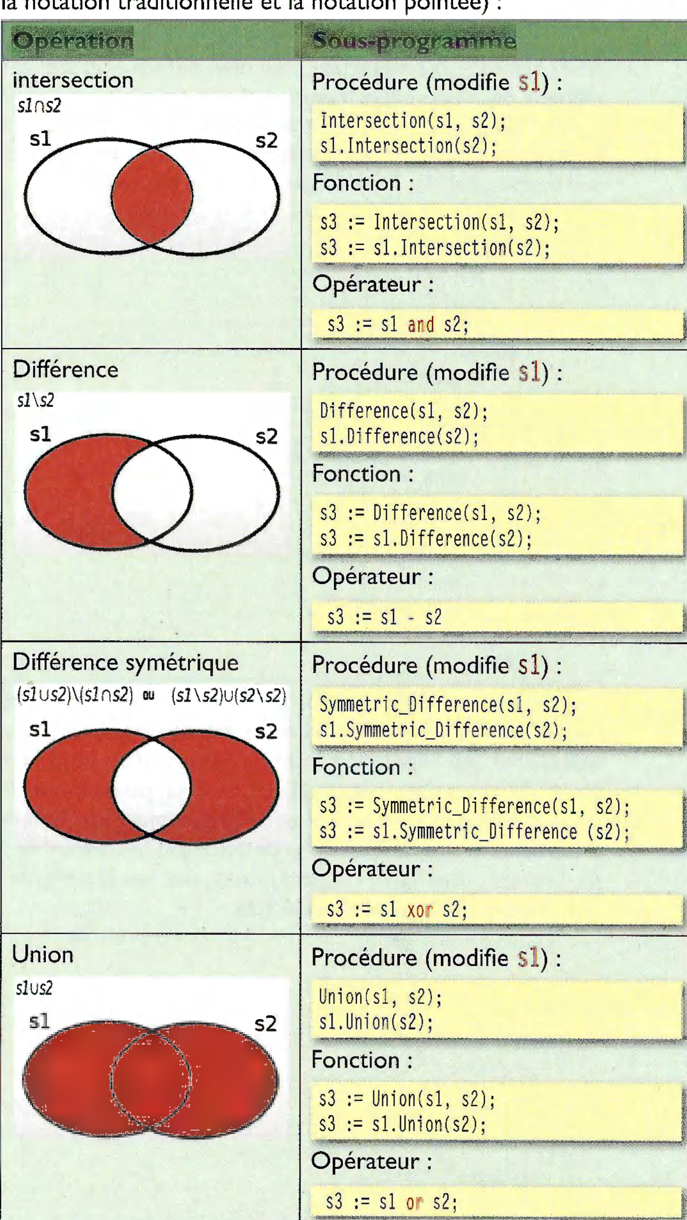
```
[ 1 -1  2 -2 ]
[-2 -1  1  2 ]
```

Vous pouvez constater qu'un ensemble issu de `Hashed_Sets` n'est pas ordonné, tandis que `Ordered_Sets` offre des ensembles ordonnés. Savoir lequel des deux types utiliser dépend de vos besoins. D'une manière générale, un ensemble s'appuyant sur une fonction de hachage sera plus efficace pour les opérations d'insertion ou de recherche pour un grand nombre de valeurs, tandis qu'un ensemble basé sur un arbre présente l'intérêt de maintenir l'ordre.

Opérations ensemblistes

Voyons maintenant quelques opérations communes. Elles sont généralement disponibles sous trois formes : une procédure modifiant son premier paramètre, une fonction de même nom retournant un nouvel ensemble, ou un opérateur surdéfini.

Si `s1` et `s2` sont deux ensembles de même type, les opérations suivantes sont disponibles (données le cas échéant en utilisant la notation traditionnelle et la notation pointée) :



Opération	Sous-programme

intersection
s1∩s2
s1 s2
	Procédure (modifie `s1`) :
	`Intersection(s1, s2);`
	`s1.Intersection(s2);`
	Fonction :
	`s3 := Intersection(s1, s2);`
	`s3 := s1.Intersection(s2);`
	Opérateur :
	`s3 := s1 and s2;`

Différence
s1\s2
s1 s2
	Procédure (modifie `s1`) :
	`Difference(s1, s2);`
	`s1.Difference(s2);`
	Fonction :
	`s3 := Difference(s1, s2);`
	`s3 := s1.Difference(s2);`
	Opérateur :
	`s3 := s1 - s2`

Différence symétrique
(s1∪s2)\(s1∩s2) ou *(s1\s2)∪(s2\s1)*
s1 s2
	Procédure (modifie `s1`) :
	`Symmetric_Difference(s1, s2);`
	`s1.Symmetric_Difference(s2);`
	Fonction :
	`s3 := Symmetric_Difference(s1, s2);`
	`s3 := s1.Symmetric_Difference (s2);`
	Opérateur :
	`s3 := s1 xor s2;`

Union
s1∪s2
s1 s2
	Procédure (modifie `s1`) :
	`Union(s1, s2);`
	`s1.Union(s2);`
	Fonction :
	`s3 := Union(s1, s2);`
	`s3 := s1.Union(s2);`
	Opérateur :
	`s3 := s1 or s2;`

Ces opérations ne peuvent s'appliquer qu'entre ensembles de même type et de même nature. Par ailleurs, la norme Ada2005 recommande (mais n'impose pas) que la complexité moyenne des opérations d'insertion, de suppression ou de recherche soit en $O(\log(N))$ pour les dictionnaires de `Hashed_Sets` ou au pire en $O(\log(N)^2)$ pour les dictionnaires de `Ordered_Sets`,

lorsqu'elles impliquent des valeurs. Ces complexités devraient par contre être en O(1) lorsque les opérations sont effectuées par l'intermédiaire de curseurs.

Les dictionnaires

Les dictionnaires permettent de réaliser l'indexation de valeurs d'un type (presque) quelconque par des valeurs d'un autre type (presque) quelconque. Les valeurs indexées sont les *éléments*, les valeurs qui indexent sont les *clefs*. Un exemple typique sont des salaires (valeurs numériques) indexés par des noms (chaînes de caractères). Un dictionnaire est comparable à un tableau, sauf que l'indice n'est pas forcément un nombre entier. Comme les ensembles que nous venons de voir, les types « dictionnaire », nommés `Map`, sont fournis par deux paquetages selon la façon dont sont organisées les clefs :

- `Ada.Containers.Hashed_Maps` pour une organisation utilisant une fonction de hachage sur les valeurs des clefs ;
- `Ada.Containers.Ordered_Maps` pour une organisation fondée sur un ordonnancement des valeurs des clefs.

On pourrait dire qu'un dictionnaire est un ensemble de paires *(clef, valeur)* n'opérant que sur la clef. Par exemple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Strings.Unbounded.Hash;
with Ada.Containers.Hashed_Maps;
with Ada.Containers.Ordered_Maps;
procedure Map_1 is
   package Salaries_Hashed is
      new Ada.Containers.Hashed_Maps(
         Key_Type        => Unbounded_String,
         Element_Type    => Integer,
         Hash            => Ada.Strings.Unbounded.Hash,
         Equivalent_Keys => "=");
   use type Salaries_Hashed.Cursor;
```

On commence par instancier un paquetage `Salaries_Hashed` fournissant un type dictionnaire indexant des entiers `Integer` par des chaînes de caractères `Unbounded_String`. Ce type, déclaré dans `Ada.Strings.Unbounded`, représente une chaîne de caractères dynamique pouvant s'étendre ou se réduire selon les besoins : il est donc beaucoup plus proche du type C++ standard `std::string` que le type usuel `String`, qui n'est en fait qu'un tableau de caractères.

La raison pour laquelle nous utilisons ce type est que dans tous les conteneurs que nous avons rencontrés jusqu'ici, les types stockés doivent être *définis*. Or, le type `String` est déclaré ainsi :

```
type String is array(Positive range <>) of Character;
```

Il s'agit d'un type tableau non contraint : en lui-même, le type `String` ne définit pas les bornes de ce tableau. On dit alors que ce type est *indéfini*. C'est également le cas pour les types « enregistrement » (`record`) présentant un discriminant (revoyez éventuellement le cinquième article de cette série, dans *Linux Magazine* 75).

De tels types indéfinis ne peuvent pas être stockés dans les conteneurs que nous avons vus jusqu'ici (mais nous verrons bientôt comment faire). C'est pourquoi nous utilisons le type `Unbounded_String`, qui est un type défini, le tableau de caractères interne étant manipulé par la mémoire dynamique.

Comme pour le type ensemble déclaré dans `Hashed_Sets`, nous devons fournir une fonction de hachage sur notre type de clefs – c'est-à-dire sur le type `Unbounded_String`. Heureusement, une telle fonction est disponible dans la bibliothèque standard, `Ada.Strings.Unbounded.Hash`. L'équivalence entre clefs est définie par l'égalité entre les chaînes de caractères.

Voyons comment écrire une simple procédure affichant le contenu d'un dictionnaire, tel que défini par notre paquetage :

```
   procedure Put_Line(m: in Salaries_Hashed.Map) is
      crs: Salaries_Hashed.Cursor;
   begin
      Put("{ ");
      crs := m.First;
      while crs /= Salaries_Hashed.No_Element
      loop
         Put("(");
         Put(To_String(Salaries_Hashed.Key(crs)));
         Put(" -> ");
         Put(Integer'Image(Salaries_Hashed.Element(crs)));
         Put(") ");
         Salaries_Hashed.Next(crs);
      end loop;
      Put_Line("}");
   end Put_Line;
```

Comme tous les conteneurs, les dictionnaires possèdent un type `Cursor` permettant de les parcourir. Sauf qu'ici un curseur pointe sur une paire *(clef, valeur)* plutôt que sur une simple valeur. La clef est donnée par la fonction `Key()`, la valeur par la fonction `Element()`. En dehors de cette petite subtilité, la procédure précédente est tout à fait similaire à celles que nous avons déjà vues.

Poursuivons avec un paquetage de dictionnaires ordonnés :

```
   package Salaries_Ordered is
      new Ada.Containers.Ordered_Maps(
         Key_Type     => Unbounded_String,
         Element_Type => Integer);
   use type Salaries_Ordered.Cursor;

   procedure Put_Line(m: in Salaries_Ordered.Map) is
...
   end Put_Line;
```

Il suffit cette fois de donner les types pour les clefs et les valeurs. La procédure d'affichage `Put_Line()` n'est pas détaillée, elle ressemble parfaitement à la précédente.

Utilisons maintenant tout cela :

```
   m_hash: Salaries_Hashed.Map;
   m_ord : Salaries_Ordered.Map;
begin
   m_hash.Insert(To_Unbounded_String("wxyz"), 10);
   m_hash.Insert(To_Unbounded_String("abcd"), 20);
   m_hash.Insert(To_Unbounded_String("mnop"), 30);
   Put_Line(m_hash);
   m_ord.Insert(To_Unbounded_String("wxyz"), 10);
   m_ord.Insert(To_Unbounded_String("abcd"), 20);
   m_ord.Insert(To_Unbounded_String("mnop"), 30);
   Put_Line(m_ord);
end Map_1;
```

L'ajout d'une paire *(clef, valeur)* dans un dictionnaire se fait par l'opération Insert().

La fonction To_Unbounded_String() convertit une chaîne de type String en une chaîne de type Unbounded_String, opération nécessaire étant donné que les écritures littérales comme "abcd" sont interprétées comme des chaînes String.

Voici enfin ce que ce programme affiche :

```
{ (wxyz -> 10) (abcd -> 20) (mnop -> 30) }
{ (abcd -> 20) (mnop -> 30) (wxyz -> 10) }
```

Comme vous pouvez le constater, on retrouve chez les dictionnaires certaines caractéristiques des ensembles : ceux fondés sur une fonction de hachage stockent leurs éléments dans un ordre quelconque, tandis que ceux fondés sur la comparaison des clefs ordonnent celles-ci.

Sans contraintes

L'exemple précédent a montré que nous ne pouvions stocker dans les conteneurs que des types définis. Cela signifie, par exemple, qu'il est impossible de déclarer une liste de chaînes de caractères de type String... du moins avec ce que nous avons vu.

Car c'est en réalité bel et bien possible. Six autres paquetages sont disponibles, dont les noms sont ceux que nous venons de voir préfixés par Indefinite_. Ils proposent les mêmes structures de données, mais pouvant contenir des types indéfinis – par exemple, des tableaux non contraints... Voici un exemple d'une liste chaînée de chaînes de caractères, sans passer par le type Unbounded_String :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Indefinite_Doubly_Linked_Lists;
procedure Indef_List is
   package String_Lists is
      new Ada.Containers.Indefinite_Doubly_Linked_
Lists(
         Element_Type => String);
   use type String_Lists.Cursor;
```

Le paquetage qui nous intéresse ici est Indefinite_Doubly_Linked_Lists. Si vous retirez les parties « Indefinite_ » dans ce qui précède, l'instanciation du paquetage sera refusée par le compilateur, car Doubly_Linked_Lists ne peut accepter que des types définis – ce que n'est pas le type String.

La procédure d'affichage de la liste n'est pas plus compliquée que précédemment :

```
   procedure Put_Line(l: in String_Lists.List) is
      crs: String_Lists.Cursor;
   begin
      crs := l.First;
      Put("[ ");
      while crs /= String_Lists.No_Element
      loop
         Put("""");
         Put(String_Lists.Element(crs));
         Put(""" ");
         String_Lists.Next(crs);
      end loop;
      Put_Line("]");
   end Put_Line;
   lst: String_Lists.List;
begin
   lst.Append("efgh");
   lst.Append("ijkl");
   lst.Prepend("abcd");
   Put_Line(lst);
end Indef_List;
```

Pour enfoncer le clou, voyons un exemple de dictionnaire plus sophistiqué. Supposons que nous devions stocker les patients d'un médecin, indexés par leur nom. Pour chaque patient, on mémorise quelques caractéristiques, maladies, etc. On pourrait commencer ainsi :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;
with Ada.Containers.Indefinite_Hashed_Maps;
with Ada.Strings.Hash;
procedure Indef_Map is
   type Sexe_Type is (Homme, Femme);
   type Patient_Type(Sexe: Sexe_Type) is
   record
      naissance: Time;
      case Sexe is
         when Homme =>
            hydrocele: Boolean;
         when Femme =>
            nb_enfants: Natural;
      end case;
   end record;
```

Le type Patient_Type est paramétré selon le sexe du patient (Sexe_Type). Selon ce sexe, des informations différentes sont stockées (si vous ignorez le sens de *hydrocèle*, disons qu'il s'agit d'un petit ennui que les femmes ne peuvent pas connaître). Un tel type paramétré est par essence un type indéfini, donc impossible à stocker dans les conteneurs que nous avons vus au début. Mais il existe un type dictionnaire indéfini :

```
   package Patients_Maps is
      new Ada.Containers.Indefinite_Hashed_Maps(
         Key_Type        => String,
         Element_Type    => Patient_Type,
         Hash            => Ada.Strings.Hash,
         Equivalent_Keys => "=");
```

Nous instancions le paquetage Indefinite_Hashed_Maps, en donnant comme type de clef le type (indéfini car non contraint) String et comme type d'élément le type (indéfini car paramétré) Patient_Type. Malgré cette particularité d'avoir clefs et valeurs de types indéfinis, les dictionnaires fournis par ce paquetage s'utilisent tout à fait naturellement :

```
   patients: Patients_Maps.Map;
begin
   patients.Insert("Untel",
                   (Sexe      => Homme,
```

```
                  naissance => Time_Of(1974, 2, 21),
                  hydrocele => False));
   patients.Insert("Unetelle",
                  (Sexe      => Femme,
                   naissance  => Time_Of(1980, 4, 3),
                   nb_enfants => 0));
end Indef_Map;
```

Il s'agit là d'un moyen puissant pour construire des structures de données hétérogènes, sans passer par une modélisation objet sophistiquée.

Étant donné les avantages manifestes des conteneurs acceptant des types indéfinis, pourquoi ne pas alors les utiliser constamment ? Et à quoi servent finalement les conteneurs « contraints » ?

La réponse tient à la performance. L'utilisation d'un type défini est plus simple, donc plus rapide, qu'un type indéfini. Prenons un exemple trivial consistant à ajouter des éléments (des entiers) à un vecteur, par l'opération `Append()`, sans réservation préalable. À chaque ajout, le conteneur doit s'agrandir pour accueillir le nouvel élément. Voici le programme tout simple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Command_Line; use Ada.Command_Line;
```

Le paquetage `Ada.Command_Line` permet de récupérer les paramètres passés par la ligne de commande.

```
with Ada.Calendar; use Ada.Calendar;
```

Dans `Ada.Calendar` est déclaré un type `Time` représentant une date, ainsi que quelques opérations pratiques pour mesurer un intervalle de temps.

```
with Ada.Containers.Vectors;
with Ada.Containers.Indefinite_Vectors;
procedure Bench is
   package Int_Vectors is
      new Ada.Containers.Vectors(
         Index_Type => Positive,
         Element_Type => Integer);
   package Indef_Int_Vectors is
      new Ada.Containers.Indefinite_Vectors(
         Index_Type => Positive,
         Element_Type => Integer);
```

On instancie les deux paquetages de vecteur, le premier prenant des types définis, le second des types indéfinis. Remarquez que ce dernier peut accepter également des types définis.

```
   nb     : Natural := 10000;
   def    : Int_Vectors.Vector;
   indef  : Indef_Int_Vectors.Vector;
   t_start: Time;
   t_end  : Time;
begin
   if Argument_Count > 0
   then
      nb := Natural'Value(Argument(1));
   end if;
   Put_Line(Natural'Image(nb));
   -- vecteur "défini"
   t_start := Clock;
   for i in 1..nb
   loop
      def.Append(i);
   end loop;
   t_end := Clock;
```

```
   def.Clear;
   Put_Line(Duration'Image(t_end - t_start));
```

On affiche d'abord le temps d'exécution (en secondes) de la boucle pour le vecteur de types définis, puis...

```
   -- vecteur "indéfini"
   t_start := Clock;
   for i in 1..nb
   loop
      indef.Append(i);
   end loop;
   t_end := Clock;
   indef.Clear;
   Put_Line(Duration'Image(t_end - t_start));
end Bench;
```

...le temps d'exécution de la boucle pour le vecteur de types indéfinis. Compilé sans aucune optimisation, le remplissage par vingt millions de valeurs nécessite (environ) 1.6s pour le premier cas, 3.6s pour le second : le vecteur « indéfini » est donc deux fois plus lent. Compilé en optimisant par `-O2`, on obtient 0.9s pour le premier cas et 3.1s pour le second : l'écart est donc encore plus important.

Moralité : prenez soin d'utiliser la « bonne » version des conteneurs en fonction de vos besoins.

Conclusion

Voilà pour cette présentation générale des structures de données disponibles dans la bibliothèque standard Ada2005. Comme toujours, de nombreux aspects ont été passés sous silence : consultez le standard du langage pour plus d'informations. Mais vous devriez déjà être capable de construire des structures amusantes, comme un dictionnaire de listes d'ensembles...

La prochaine fois, nous reviendrons sur les types fondamentaux que sont les caractères et les chaînes, que nous utilisons largement sans vraiment connaître leurs possibilités.

Yves Bailly,

http://www.kafka-fr.net

RÉFÉRENCES

- **[1]** Booch Components : **http://booch95.sourceforge.net**
- **[2]** Simple Components : **http://www.dmitry-kazakov.de/ada/components.htm**